Python Printing, Input, Scripts, and Functions

Main Points

1. Remember the `print()` function?
2. Functions vs. methods, fight!
3. Printing with *f-strings* to create formatted output
4. Requesting user input for interactive scripts
5. Create, edit, and execute simple scripts
6. Recall that spaces are important
7. Create and execute your own custom functions

1 The `print()` Function... Again

The `print()` function can be used to display text strings, numeric values, or a combination of the two.

- Multiple items of the same or different types can be printed by separating them with commas
- Multiple strings can be combined by adding (concatenating) them with the + operator
- Multiplying a string by an integer in `print()` will print the string that number of times
- Escape sequences can be added to strings that perform specific duties...
  - Force a line return by adding the newline escape sequence `\n`
  - Add a tab with the `\t` escape sequence

Review of `print()`

```
print("Python is awesome")
print(4*5)
print('Lumberjacks', "Parrots", 42)
print(2*'Hello')
print('Hello,\nWorld!')
print('Hello\tthere,\tagain')
```

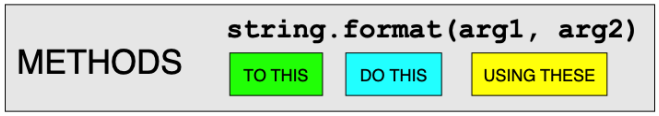
```
Python is awesome
20
Lumberjacks Parrots 42
HelloHello
Hello,
World!
Hello	there,	again
```

2 Functions Versus Methods

Python uses both functions and methods to work with/on objects. The included images are meant to describe the general format and operation of both.

Functions perform an operation on or with with zero or more arguments.

- Arguments (also called parameters) are values placed within the parentheses of a function
- Most functions return a value or values
  - `abs(-100)` returns the absolute value of `x`
  - `divmod(x, y)` returns both the quotient and remainder of $x \div y$
- Some functions, like `print()`, perform an operation but do not return any values



Methods work on specific object types to return a value or change the object.

- Methods can accept arguments but often don't
- Most methods that don't accept arguments still require a set of parentheses
- Syntax is slightly different than for functions; the object is followed by a dot, the method name, and parentheses (with or without arguments)
- The `str.format()` method can be used to format portions of a string
- Use `dir(object_name)` for any object type to list available methods, i.e. `dir(str)` will list the string methods (plus more)

Printing the directory for strings - methods only

```
>>> print([item for item in dir(str) if not item.startswith('__')])
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 ↪  'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 ↪  'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
 ↪  'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
 ↪  'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 ↪  'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
 ↪  'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 ↪  'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> "Hello".upper()
'HELLO'
>>> "Hello".startswith("H")
True
```

3 Formatting Printed Output

3.1 Using the `.format()` String Method

Including the `.format()` string method within a `print()` function allows for controlling exactly how numeric (and non-numeric) values are displayed. The general layout of the `.format()` method is as follows...

```
"The sum of {} and {} is {}".format(item_1, item_2, item_3)
```

- The expression starts with a string that has curly braces `{}` as placeholders
- The curly braces act as blanks in a fill-in-the-blanks statement
- The `.format()` method directly follows the closing quote for the string
- The arguments in the parentheses are values or expressions (in order) that match up with the placeholders in the string
- Placeholders can include formatting descriptors to generate specific formatting
  - Formatting descriptors must be preceded by a colon, i.e. `{:.2f}`
  - The website https://pyformat.info has many examples of the `.format()` method; from simple to complex

The result of 22/7 is an estimate for π that has historically been used by many tool makers and machinists. The following examples use `print()` and the `.format()` string method to display the result of 22/7.

No `.format()` – just using `print()` with 2 arguments and a comma

```
print('pi is close to', 22 / 7)
```

```
pi is close to 3.142857142857143
```

Using `.format()` with `{}` – no specific formatting assigned

```
print('pi is close to {}'.format(22/7))
```

```
pi is close to 3.142857142857143
```

Using `.format()` with `{:f}` – floating point notation with default decimal places

```
print('pi is close to {:f}'.format(22/7))
```

```
pi is close to 3.142857
```

Using `.format()` with `{:.4f}` – floating point notation with 5-decimal places

```
print('pi is close to {:.4f}'.format(22/7))
```

```
pi is close to 3.1429
```

3.2 Formatted String Literals – The New Way

Formatted string literals, also called "*f-strings*", were added to *Python* starting with version 3.6. They work like the `.format()` method but in a more direct way. The general layout of the *f-strings* is as follows...

```
"The sum of {item_1} and {item_2} is {item_3}"
```

- *F-strings* allow expressions or variables to be placed directly within the curly braces
- Adding a colon after the expression or variable to add a formatting descriptor, for example `f"pi is close to {22/7:.8f}"`

F-string with `:f` – floating point notation with default decimal places

```
print(f'pi is close to {22/7:f}')
```

```
pi is close to 3.142857
```

F-string with `:.16f` – floating point notation with 16-decimal places

```
print(f'pi is close to {22/7:.16f}')
```

```
pi is close to 3.1428571428571428
```

F-string with `:e` – exponential notation (`:E` for uppercase)

```
print(f'pi is close to {22/7:e}')
```

```
pi is close to 3.142857e+00
```

F-string with `:.12E` – exponential notation with 12-decimal places

```
print(f'pi is close to {22/7:.12E}')
```

```
pi is close to 3.142857142857E+00
```

F-string with `:g` – more efficient of standard or exponential notation (`G` for uppercase)

```
print(f'pi is close to {22/7:g}')
```

```
pi is close to 3.14286
```

F-string with `:.12g` – 12-decimal places, standard or exponential notation

```
print(f'pi is close to {22/7:.12g}')
```

```
pi is close to 3.14285714286
```

F-string with `:8.3f` – total width of 8 characters with 3 to the right of the decimal

```
print(f'pi is close to {22/7:8.3f}')
```

```
pi is close to    3.143
```

Breaking down the `:8.3f` example...

- 8 => 8 total characters set aside for the value
- .3 => 3 digits to the right of the decimal point
- f => the value will be formatted as a float
- The decimal point counts as a character
- Leading blanks are added as needed

```
| | | |3|.|1|4|3|  <= formatted value
| | | | | | | | |
|8|7|6|5|4|3|2|1|  <= characters set aside for the value
```

F-string with `:+08.3f` – Same as the previous but with leading zeros and the + or - sign

```
print(f'pi is close to {22/7:+08.3f}')
```

```
pi is close to +003.143
```

`F-string with :.0f` – Force a float to display no values right of the decimal (like an integer)

```
print(f'pi is close to {22/7}')
```

```
pi is close to 3.142857142857143
```

Using a variable in an f-string

```
almost_pi = 22/7
print(f'pi is close to {almost_pi:.8f}')
```

```
pi is close to 3.14285714
```

`:d` – Standard integer (object must be of `int` type)

```
print(f'Integer value: {42:d}')
```

```
Integer value: 42
```

Formatting for dollars and cents

```
print(f'${1000000:,.2f} is not as much as it used to be')
```

```
$1,000,000.00 is not as much as it used to be
```

`:4d` – Integer with 4 total characters

```
print(f'Integer value: {42:4d}')
```

```
Integer value:   42
```

`:04d` – Integer with 4 total characters and leading zeros

```
print(f'Integer value: {42:04d}')
```

```
Integer value: 0042
```

`:+04d` – Integer with 4 total characters and leading zeros and the $\pm$ sign

```
print(f'Integer value: {42:+04d}')
```

```
Integer value: +042
```

`:,d` – Integer with comma separators

```
print(f'Integer value: {987654321:,d}')
```

```
Integer value: 987,654,321
```

:s String (although setting the formatting is not really necessary in this case)

```
first_name = "Slim"
last_name = "Shady"
print(f'I am the real {first_name:s} {last_name:s}')
```

```
I am the real Slim Shady
```

:^20s – String centering in space 20 characters wide (use < and > for left/right justified)

```
first_name = "Slim"
last_name = "Shady"
print(f'I am the real {first_name:^20s} {last_name:^20s}')
print(f'I am the real {first_name:<20s} {last_name:>20s}')
```

```
I am the real         Slim                Shady
I am the real Slim                                Shady
```

4 The `input()` Function

Using the `input()` function to request information from a user within a script will make the script interactive and allow the user to run the script with values of their choosing.

- `input()` accepts one optional argument
  - A statement or question to the user so they know what to enter
  - Must be a string or formatted string
- The return value from `input()` is always a string
  - Must specifically convert the returned value to an integer or float if that is desired
  - The last two examples below have the `input()` function inside of `float()` and `int()`
- Good practice: end prompt strings with a delimiter of some type followed by a space
- Common delimiters
  - Question mark (?)
  - Colon (:)
  - Right arrow (>)
- The delimiter and space helps readability, so please *do it*

Sample `input()` statements

```
user_name = input('What is your name? ')
city = input("Enter your city of residence: ")
applied_load = float(input('Enter the applied load (lbf): '))
age = int(input('Enter your current age > '))
```

5 *Python* Scripts

Most of the time it is more efficient to use a script (or user-defined function) than to enter commands at a REPL prompt or in a *Jupyter* code cell. A script can be used on any computer with *Python* installed. Functions are usually written as part of a script or module. When a function is written within a script it is called/used in the script's code. A script is referred to as a module if it just contains functions for importing and using in other scripts. Calling functions that are in modules requires importing the module file then calling the function (see the examples below).

Running a script from a *Python* prompt

```
>>> import hello
Hello, World!
```

Importing a module and calling a function from a *Python* prompt

```
>>> import my_module
>>> my_module.my_function()
Nice function
```

Importing and calling a specific function from module

```
>>> from my_module import cuberoot
>>> cuberoot(27)
3.0
```

5.1 Script Details

Scripts are sometimes referred to as programs. Simple scripts are text files with lines of commands/expressions that are executed sequentially from top to bottom.

- Scripts may receive necessary values using several methods
  - Assign to variables directly within the script ("hard-coding")
  - Ask the user to enter values at a prompt using the `input()` function
  - Pass values to the script when it is executed (not covered here)
  - Load values from a file (in a future document)
- Variables in scripts are only available for use after being assigned
- Results from scripts can be made available to the user via...
  - The `print()` function
  - Writing output values to a text file (in a future document)

5.2 Good Programming Practices

Try to use good programming practices when writing scripts. Doing so makes them easier to debug and for others to read at a later date. Remember that code is read more often than it is written.

Good practices include (but are not limited to) the following:

- Write an informational docstring-style comment at the top of the script
  - Describing what the script does
  - Units that are used, special conditions, etc
  - Author's name, website, licensing, etc
- Place all `import` statements near the beginning of script after the initial comment
- Place all user-defined function definitions immediately after imports
- Use descriptive variable names
- Include comments throughout the script explaining "why" things are being done
- Clean up your code with a style formatter such as *Black* or *autoPEP8*

5.3 Editing Scripts

- *Python* scripts must be written in plain text
  - Good - a stand-alone text editing applications
  - Better - an editor built into a *Python* capable integrated development environment (IDE)
- Word processors like *Word* should never be used for writing or editing scripts
- *Python* installations have a built-in text editor/IDE named *IDLE*
- Dedicated coding text editors, like *Visual Studio Code* (*VS Code*), often add nice features; including the ability to run scripts directly from the editor
- Good, beginner-friendly IDEs
  - *Thonny* - can install and use a "local" version of *Python* from within the application
  - *Mu* - possibly the best IDE for *CircuitPython*

5.4 Naming scripts

- No spaces in the name
- Must start with a letter (or an underscore character, but this is infrequent)
- May include only letters, numbers, and underscores
- Should be descriptive of what the script is used for
- Must end with `.py` file extension
- Some valid script names...
  - `stress_calculator.py`
  - `fibonacci.py`
  - `_calculator.py`
  - `easy_as_123.py`
  - `uscs2si.py`

5.5 Running Scripts

- From a console command line prompt such as Windows Power Shell or the MacOS Terminal app
  - Type `python` or `python3` followed by the script name, including the `.py` extension
  - Examples...
    - `python hello.py`
    - `python3 hello.py`

Running a script from console prompt

```
% python hello.py
Hello, World!
```

- From a *Python* prompt
  - Change to the directory containing the script file before starting *Python*
  - Start *Python*, type `import script_name` at the >>> prompt, and press [return]

Running a script from a *Python* REPL prompt

```
>>> import hello
```

5.6 Example Scripts

5.6.1 Script with hard-coded values

This script converts from mph to km/h. The value of `mph` will be "hard-coded" instead of using an `input()` function. The script is saved to a text file named `mph2kph.py`. It is important that there are no spaces before, after, or within the script name and that the extension is included.

Python script – `mph2kph.py`

```
"""
this script converts mph to km/h
the conversion factor is 1 mph = 1.609 km/h

author: Brian Brady
        Ferris State University
"""

mph = 60                # speed in mph is hard-coded to 60 for now

conversion = 1.609      # conversion factor from miles to km

kph = mph * conversion  # kph is the same as km/h

# 60 mph is 96.53999999 km/h
print(f"{mph} mph is {kph} km/h")

# end of script
```

```
60 mph is 96.53999999999999 km/h
```

5.6.2 Adding `input()` to the Previous Script

The previous script can be improved with interactive user input. Instead of assigning a fixed value to `mph`, an `input()` function can be used to ask the user to enter a speed in mph. Placing the `input()` function inside the parentheses of a `float()` function will convert the input value from a string into a float so calculations can be performed with the input value.

The `print()` expression has been modified to use formatted output that displays the input speed with no special formatting and the calculated speed with one decimal place.

Python script with user input – `mph2kph_input.py`

```
"""
this script converts mph to km/h
user enters mph when prompted
the conversion factor is 1 mph = 1.609 km/h

author: Brian Brady
        Ferris State University
"""

mph = float(input("Enter a speed in mph: "))
conversion = 1.609
kph = mph * conversion

print(f"{mph} mph is {kph:.1f} km/h")

# end of script
```

5.6.3 An Example Script with Multiple Inputs

The following script named `windchill.py` uses `input()` functions to ask the user to enter the air temperature in degrees F and the wind speed in mph. It then calculates the wind chill temperature T_{wc} in degrees F. Following is the formula for calculating the wind chill temperature T_{wc}.

$$T_{wc} = 35.74 + 0.6215\,T - 35.75\,v^{0.16} + 0.4275\,T\,v^{0.16}$$

The statement `"xxx F with a yyy mph wind equals a zz.z F wind chill"` will be displayed using formatted printing such that the input values have no special formatting and the calculated value is a float with one decimal place. An introductory docstring-style comment describes what the script does, the units being used, and the author's name (similar to the previous scripts). Testing the script with the following values will yield the results shown.

- $30°\text{F}$ with 10 mph $\Longrightarrow 21°\text{F}$
- $10°\text{F}$ with 30 mph $\Longrightarrow -12°\text{F}$
- $-10°\text{F}$ with 20 mph $\Longrightarrow -35°\text{F}$

Python script – `windchill.py`

```
"""
Determines windchill in degrees F
Inputs are air temperature in degrees F and wind speed in mph

Author: Brian Brady, Ferris State University
"""

T = float(input("Enter air temperature in F: "))
v = float(input("Enter wind speed in mph: "))

Twc = 35.74 + 0.6215*T - 35.75*v**0.16 + 0.4275*T*v**0.16

print("{T} F with a {v} mph wind equals a {Twc:.1f} F wind chill")
```

6 Importance of Indentation

Indentation (spaces at the beginning of lines) is important in *Python*. The most important thing to remember about indentation is that it must be consistent. Spacing between objects in a line of code is generally not important however.

- In the previous script examples all lines were aligned on the left edge
- Indentation is used in *Python* to group commands for specific purposes
- The default indentation is no indentation at all
- Even a single extra space at the beginning of a line will generate an error (see below)
- Standard indentation is 4 spaces (when indentation is required)
- Spaces and tabs for indentation must not be mixed; spaces are preferred

Indentation is important

```
>>> print("This line is fine")
This line is fine

>>> # The following line has a space at the beginning
>>>  print("The space at the start of this line causes an error")
  File "<stdin>", line 1
    print("The space at the start of this line causes an error")
    ^
IndentationError: unexpected indent

>>> # Spaces between objects are ignored
>>> 25 / 5       + 2       *3
11.0
```

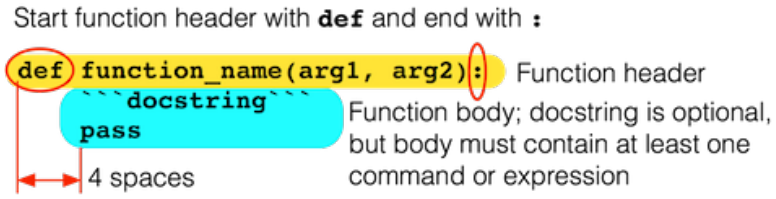
7 User-Defined (Custom) Functions

7.1 User-Defined Function Details

- Contained within a script or a module file
- Must be defined before they can be called (used)
- Usually receive input by passing arguments instead of using `input()` commands
- `abs(-100)` *passes* `-100` as an argument to the absolute value function
- Usually return results back to the calling location instead of printing the result
- `abs(-100)` *returns* the value `100` but does not print the result
- Argument names passed into and used in a function are only available for use in that function; it is said they are "local" to the function
- Variables created inside a function are only available inside that function (local variables)
- Can be written to return results, print results, or return and print results
- Functions that don't return values are sometimes called void functions
- Functions that do return values are sometimes called fruitful functions

7.2 User-Defined Function Structure

- Indent all commands that belong to the function by 4-spaces except the header
- Indenting signals to *Python* that the lines belong to the function definition



- The first line is the function header and includes...
  - The command `def`
  - Function name
  - A set of parentheses
  - Any arguments to pass (if any) inside the parentheses
  - A colon at the end of the line
- Lines below the header are the function body
  - Must consist of at least one command
  - The `pass` command may be used as a placeholder if there are no other commands
  - A `pass` command tells *Python* to exit the function and go back to where it was
- Function definitions may optionally include a docstring
  - Displays information about the function when `help()` is called with the function name
  - The docstring must be enclosed by a set of three double quotes
  - Can span multiple lines if needed, but just one set of double quotes total
  - The docstring must start on the line immediately below the header

Defining a function and then calling it

```
def do_nothing():
    """
    This function does nothing useful.
    Arguments: no arguments accepted
    Prints: "Nothing done"
    Returns: does not return anything
    """
    print("Nothing done")
```

```
>>> do_nothing()
Nothing done

>>> help(do_nothing)
Help on function do_nothing:

do_nothing()
    This function does nothing useful.
    Arguments: no arguments accepted
    Prints: "Nothing done"
    Returns: does not return anything


>>> something = do_nothing()
Nothing done

>>> print(something)
None
```

7.3 Void Functions

- Void functions do not return any values
- May print something, but don't return anything
- The above function, `do_nothing()`, is a void function

7.3.1 Void Function with One Argument

The following code block contains another void function that accepts a single argument and prints the result of the argument multiplied by 2.

Void function – `double_me(value)`

```
def double_me(value):
    """
    This function multiplies the argument named value by 2
    and prints the result
    """
    doubled_value = value * 2
    print(doubled_value)
```

```
>>> double_me(10)
20
```

7.3.2 Void Function with Multiple Arguments

This next example is also a void function, but it accepts two arguments. Notice that the argument names used in this and the previous example are not the same. You can use any valid variable name as a function argument name. Also note that the calculation is performed directly in the `print()` function.

Void function – `multiply2(arg1, arg2)`

```
def multiply2(arg1, arg2):
    print(arg1 * arg2)
```

```
>>> multiply2(6, 7)
42
```

7.3.3 Void Function with Formatted Printing

The following example defines and calls a function named `print_mph2kph` that accepts one argument named `mph`. The function converts `mph` to km/h and assigns the result to the name `kph`. On the last line of the function the result is printed such that it reads `"xxx mph equals yyy km/h"`, where `xxx` and `yyy` are replaced by the values of `mph` and `kph`.

Void function – `print_mph2kph(mph)`

```
def print_mph2kph(mph):
    """calculate and print km/h from mph"""
    kph = mph * 1.609
    print(f"{mph} mph equals {kph} km/h")
```

```
>>> print_mph2kph(60)
60 mph equals 96.53999999999999 km/h
```

7.4 Fruitful Functions

- Fruitful functions return a value or values back to the caller
- They usually do not create any printed output
- Must have a `return` statement
- The `return` is usually located on the last line since the function exits once a value is returned
- Return multiple values by separating them with commas after the `return` statement
- `return` is a statement, not a function, and does not use parentheses
- The example below illustrates the general structure of a fruitful function

General structure of a fruitful function

```
def my_function(arg1, arg2):
    """docstring"""
    body line 1
    body line 2
    return value1, value2
```

7.4.1 An Example Fruitful Function

The following function definition for `double_me2` is a modification of the `double_me` function from earlier. This time, instead of printing the result of the calculation, the function returns the result.

Fruitful function – `double_me2(value)`

```
def double_me2(value):
    """
    This function multiplies the argument named value by 2
    and returns the result
    """
    doubled_value = value * 2
    return doubled_value
```

```
>>> double_me2(10)
20
>>> two_times = double_me2(21)
>>> print(two_times)
42
```

A more *Pythonic* way to define the above function would be to move the calculation into the `return` line. Unless the intermediate variable is needed for another calculation in the function, it is more efficient to just return the calculation directly.

Fruitful function – `double_me3(value)`

```
def double_me3(value):
    """
    Doubles value and returns the doubled result
    """
    return value * 2
```

7.4.2 Fruitful Function with 2 Arguments and 2 Returned Values

The following `rectangle(width, height)` function returns the area and perimeter of a rectangle with sides of `width` and `height`. The two returned values can be assigned to two variables when the function is called like so...

```
(area, perim) = rectangle(width, height)
```

The grouping `(area, perim)` is referred to as a *tuple* by *Python* and is the *Pythonic* way of assigning multiple values to multiple variable names at the same time.

Fruitful function – `rectangle(width, height)`

```
def rectangle(width, height):
    """
    Returns the area and perimeter (in that order) for a
    rectangle of width and height
    """
    area = width * height
    perimeter = 2 * width + 2 * height
    return area, perimeter
```

```
>>> rectangle(5, 8)
(40, 26)

>>> area, perim = rectangle(10, 24)

>>> print(f"Area = {area} and perimeter = {perim}")
Area = 240 and perimeter = 68
```

7.4.3 Another Fruitful Function

The previously created `print_mph2kph` function has been recreated below such that it returns the speed in km/h but does not print anything.

Fruitful function – `return_mph2kph(mph)`

```
def return_mph2kph(mph):
    return mph * 1.609
```

```
>>> return_mph2kph(55)
88.495

>>> speed = return_mph2kph(75)
>>> print(speed)
120.675
```

7.4.4 Yet Another Fruitful Function

The following function named `windchill(tempF, vel_mph)` does the same thing as the script from earlier called `windchill.py` except instead of using `input()` functions to get the air temperature and wind speed it uses two arguments named `tempF` and `vel_mph`. The resulting wind chill temperature is rounded to zero decimal places when it is returned.

$$T_{wc} = 35.74 + 0.6215\,T - 35.75\,v^{0.16} + 0.4275\,T\,v^{0.16}$$

Testing the function with the following values results in $15°\mathrm{F}$ and $-4°\mathrm{F}$.

- $30°\mathrm{F}$ with $30\mathrm{mph}$
- $10°\mathrm{F}$ with $10\mathrm{mph}$

Fruitful function – `windchill(temp_F, vel_mph)`

```
def windchill(temp_F, vel_mph):
    """
    Calculates and returns windchill temperature in degrees F
    Arguments: temp_F = air temperature in degrees F
               vel_mph = wind speed in mph
    """
    T = temp_F
    v = vel_mph
    T_wc = 35.74 + 0.6215*T - 35.75*v**0.16 + 0.4275*T*v**0.16
    return round(T_wc)
```

```
>>> windchill(30, 30)
15
>>> windchill(10, 10)
-4
```

8 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapters 3 and 6
- *The Coder's Apprentice*, Pieter Spronck, chapters 5 and 8
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 1, 10, 13, and 23
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 3 and 4